Spring 1-1-2012

Math on a Sphere: Implementing a Programming Language for Learners

Michelle Bourgeois Redick
University of Colorado at Boulder, tiggons@gmail.com

Math on a Sphere: Implementing a Programming Language for Learners

by

Michelle B. Redick

B.S., University of Colorado, 2007

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science

2012

This thesis entitled:
Math on a Sphere: Implementing a Programming Language for Learners
written by Michelle B. Redick
has been approved for the Department of Computer Science

Michael Eisenberg

Prof. Andrzej Ehrenfeucht

Prof. Elizabeth Jessup

Prof. Clayton Lewis

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Redick, Michelle B. (M.S., Computer Science)

Math on a Sphere: Implementing a Programming Language for Learners

Thesis directed by Prof. Michael Eisenberg

Science on a Sphere (SOS) is an increasingly common display at museums and planetariums all over the world. The SOS system consists of a large spherical surface about six feet in diameter upon which images and movies are displayed using four surrounding projectors. The three-dimensional surface provides a unique viewing experience to help visitors understand data such as global weather patterns, ocean currents, and mappings of the Martian surface.

However, the current utilization of this device is lacking in interactivity. Some installations allow visitors to rotate and spin the displayed image, or select different data to view, but users cannot create their own images to display. This is where Math on a Sphere (MOS) comes in. MOS is a drawing program applied to a spherical surface with the aim of interesting the public in mathematical and programming concepts, engaging visitors at science centers and museums, and providing a new medium for creative expression.

In the spirit of making this software easily accessible, our implementation is almost entirely web-based. In fact, the web application functions independently, allowing users to edit and test their programs while seeing the effects on a preview of the sphere, making the SOS hardware optional. The main focus of this thesis is the programming language developed specifically for the MOS project. This language was designed in the image of the Logo programming language due to its learner-targeted characteristics.

Dedication

To my husband Tyler, for patiently supporting me through my never-ending college career. And to my friends and family for helping to make me who I am today.

Acknowledgements

I would like to thank my advisor, Michael Eisenberg, for his dedicated interest in this project. I would also like to thank Antranig Basman for his much appreciated guidance and advice throughout the implementation of this project. This work was supported by the National Science Foundation grant #DRL1114388.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

This thesis is part of the NSF funded project "Math on a Sphere: An Interactive Exploration of 3D Surfaces for Public Audiences", a joint investigation between the University of Colorado at Boulder (CU Boulder) and the University of California Berkeley's Lawrence Hall of Science (LHS) [11]. The proposed two-year project plans to develop and evaluate a system for engaging the public in mathematics by interacting with a large spherical display, Science on a Sphere (SOS). By "opening" this large beautiful display to the public, I hope to more deeply engage museum visitors and young learners in mathematics, computer programming and creativity.

Math on a Sphere (MOS) is an interface that allows users to write programs that draw on the SOS display. The interface can be accessed online using a web browser and connects to the SOS hardware. In the browser view, users can compose code and see a preview of the output on a three-dimensional model of the sphere. When connected to the server, the images generated by the program are displayed on the six-foot sphere in near real-time. In the spirit of making this software accessible to all, we implemented a programming language with a Logo-like syntax, while providing support for some higher-order functions.

Work for the Math on a Sphere system was completed by Antranig Basman and myself since fall of 2011, with most of my work going into the implementation of the programming language. For this reason, the following paper will focus on the development of the programming language while providing an overview of the Math on a Sphere system as a whole.

1.1 Involving the Public

Science on a Sphere (SOS) was created and developed by the National Oceanic and Atmospheric Administration (NOAA) by Dr. Alexander MacDonald. These spheres are becoming increasingly common features at museums, science centers, and zoos, with over fifty installations in the United States and eighty world-wide[21], with more exhibits established each year. The system uses four projectors to display images and movies onto the surface of a sphere about six feet in diameter. This three-dimensional surface provides a unique experience for visitors to view (e.g., global weather patterns, ocean currents, mappings of the Martian surface).

Figure 1.1: The Science on a Sphere (SOS) installation at the Meteo World Pavilion in China

As beautiful as these displays are, the utility of SOS is severely lacking in interactivity. An evaluation of SOS found that the content on SOS was either too advanced or had no interactivity for children other than looking at the sphere[17]. Some installations allow visitors to rotate and spin the displayed image, or select different data to view, but there is no creative aspect to this interaction. Adding a creative element to the exhibit will truly engage the user with a "look what I did!" mentality. This gives museum and science center visitors the opportunity to take pride in their contribution, connecting on a personal level with the exhibit and the institute that acts as its caretaker.

1.2 Education

From personal experience, I know that if I am engaged in a task I will be more driven to learn the skills required to achieve that task. The same applies to many others. Therefore, providing a system that is both aesthetically appealing and intellectually challenging could lead to a greater desire to learn. Using a system to draw on a sphere would demand computational ideas when planning how to construct a drawing from the given set of tools, structure solutions, debug, and optimize[5, 19], skills that have been identified as missing components of education[4].

A spherical drawing interface would demand some knowledge of spherical geometry. In some ways, geometry is a gateway course to higher math and engineering careers such as geosciences, aerospace, astronomy, and video game development. As an example of how a user would be introduced to spherical thinking, consider a turtle crawling on the surface of the earth. In Abelson and diSessa's book *Turtle Geometry*:

> Imagine that a turtle is crawling on a sphere ...the earth's surface, for example... Starting at the equator and facing north, the turtle goes straight north until it reaches the North pole. There it turns 90° and goes straight south until it gets to the equator. Again it turns 90° and runs along the equator to get back to its initial position, where a final 90° turn restores its initial heading. [1], p. 202

This three-right-angle triangle shows right away how concepts in spherical geometry differ from Euclidean geometry.

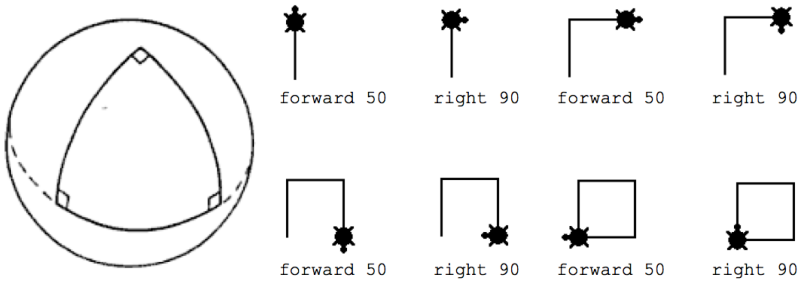


Figure 1.2: To the left, the classic turtle from the Logo programming language walks the sphere to form a triangle from three right-angles [1]. On the right, a similar path is carried out on a plane for comparison[13].

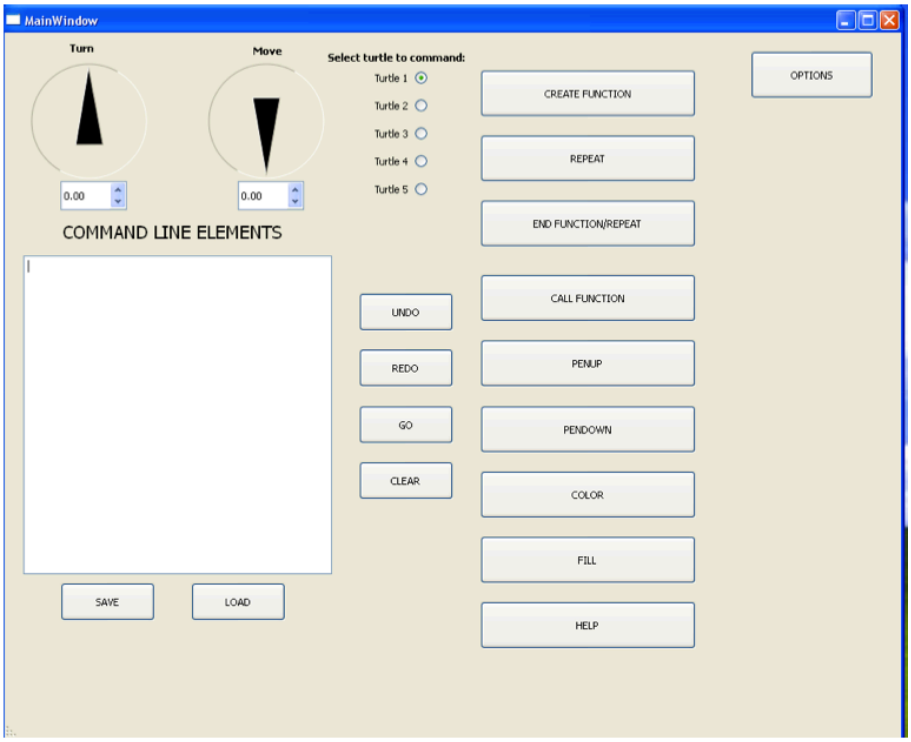
1.3 Prior Work

In 2010 a proof-of-concept project was implemented by CU Boulder students under Professor Eisenberg as a senior project assignment[10]. The user interface consisted of an editor window and several buttons used to populate the editor window with commands. The full list of commands in the editor could then be executed to generate a series of images to be displayed on the SOS system.

However, the end product required a significant amount of package installations and configuration steps to get the application up and running. For someone new to programming (and possibly computer systems in general) this could pose a daunting task. Even though the prototype did an excellent job of showing some key aspects of the Logo language, there were missing components to any programming language, like scoping, nested functions and conditionals. Additionally, the prototype was not directly connected to the SOS system. One could not simply execute a command and expect the sphere to display the result. The full program had to be compiled, converted into a movie and then transferred to the SOS system to be played on the sphere.

The system described in this paper takes a different approach. The application is almost entirely web-based, so anyone with internet access can use it. This significantly simplifies the initial setup required for a user to get started - they would just need to install a WebGL capable web browser (like the current version of Firefox or Chrome). The new design allows a user to interact with the sphere in near-real time, instead of completing the multi-step process needed for displaying programs from the old system. Finally, the focus of this paper is on the programming language, which was described using a context-free grammar, greatly increasing the extensibility of the language. The new language supports nested procedures, proper scoping for variables and functions, mathematical expressions and conditionals.

Figure 1.3: User interface for the MOS project prototype.[18]



Chapter 2

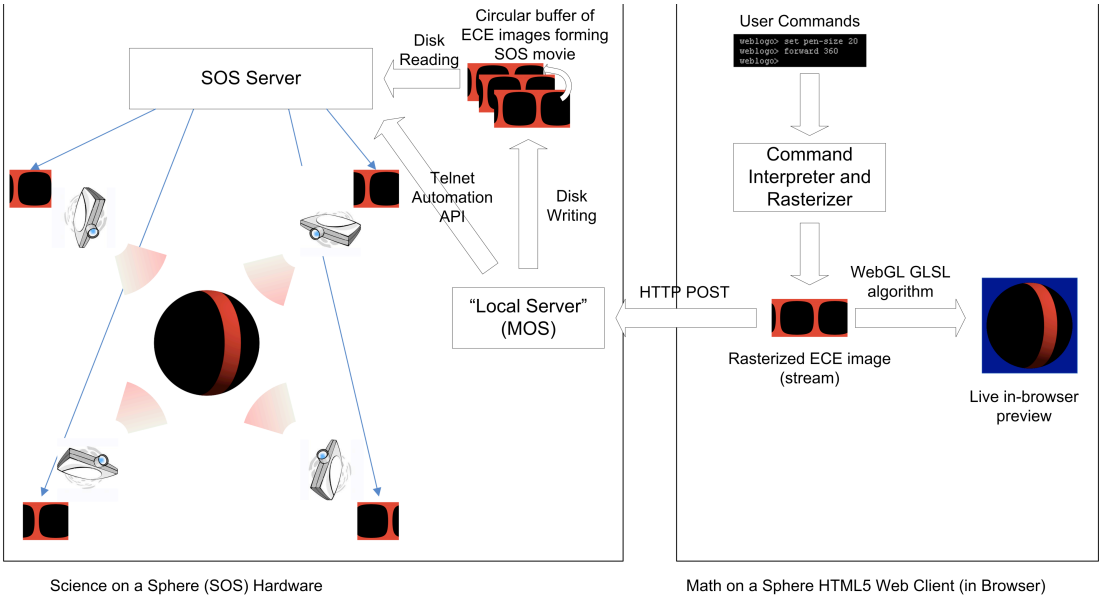
Overview of the Math on a Sphere System Design

The design of Math on a Sphere consists of two main components: a web client which serves as the user interface, and a server running on the SOS hardware. The web client is responsible for interpreting the user's input program and rasterizing the ECE (Equatorial Cylindrical Equidistant) images. Those images are then wrapped around a model of a sphere to show as a preview in the browser and are also posted to the MOS server running on the SOS hardware. The SOS hardware consists of the sphere, the four projectors, and the SOS machine responsible for managing the images sent to each projector.

The second MOS component is a server running on the SOS machine that takes the images it receives from the web client and writes them to disk in a circular buffer on the SOS machine. The MOS server establishes a Telnet connection with the SOS Server to initiate reading from the MOS image buffer, and allows the MOS server to track the progress of SOS as it displays each frame in the buffer so that the circular buffer can be properly managed.

Together, these components allow a user to interact in near-real time with the spherical display. However, the web client can be used independently from the server to experiment with programming doodles even in the absence of a Science on a Sphere installation. The design of these components is described in greater detail in this chapter, and should give an overall understanding of how the system works as a whole.

Figure 2.1: This is an overview of the Math on a Sphere system design. Two main components of MOS were developed. One is a web client in which the user interacts, while the other acts as a server on the SOS hardware to transmit images to the display. On the right is the web client; responsible for interpreting the user's input program, rasterizing the ECE images generated by that program, using WebGL to wrap the images around a 3D model to preview in the browser and posting the images to the server on the SOS system (the second component). On the left is a depiction of the SOS hardware. It consists of the sphere, the four projectors, the SOS machine running the SOS server responsible for managing the images sent to each projector. The MOS server runs on the SOS machine and takes the images it receives from the web client's post, writing them to disk in a circular buffer. The MOS server establishes a Telnet connection with the SOS Server to initiate reading from the MOS image buffer. This Telnet connection also allows the MOS server to track the progress of SOS as it displays each frame in the buffer, so that the circular buffer can be properly managed. (Image provided by Antranig Basman.)



2.1 Web Client

The web client includes the bulk of the software implementation. This includes the user interface, the interpreter for the user's commands, and the rasterizer that generates images for the SOS sphere, as well as for the WebGL preview seen in the web browser interface. These components all run on the client side and can function independently from the SOS system. This allows users to experiment with the software even if they do not have access to a sphere.

2.1.1 User Interface

The user interface is accessed though any web browser with WebGL support. I currently use Firefox 11.0 and Chrome 18.0 on a Mac, but the MOS system has been tested on other platforms as well. The design of this interface has evolved throughout the project. However, the basic components have remained the same: a command line window (where a single command may be executed at a time), an editor window (where multiple lines of code can be composed and executed as a script), a 2D preview window (where the image appears in its "flat" form), and a 3D preview window (where a model displays the images wrapped around a sphere along with the "turtle"). See figure 2.2.

On the top left of the web browser interface is the editor window, constructed using an HTML `textarea` element along with CodeMirror. CodeMirror is a JavaScript library used to create an interface for editing code within a web browser. This is where users can compose and edit multiple lines of code to execute all at once. It comes with autocomplete and syntax checking for many well-known languages such as C++, JavaScript, Java, Perl, Python, etc.[16]. This feature is not currently used, but could be modified to support our WebLogo syntax later down the road.

Just below the editor window is the command-line interpreter. This uses JQuery's Terminal emulator plugin allowing us to parse the user's input for interpretation. This is where the user could enter one command at a time. The key difference between commands executed in the command-line vs. the editor window, is that the state of the drawing and the "turtle" are reset before executing

the code in the editor, while commands executed in the command-line are cumulative, allowing the user to see how each command entered affects the state of the "turtle". This terminal emulator also gives us a convenient and logical location to print debugging information for the user[22].

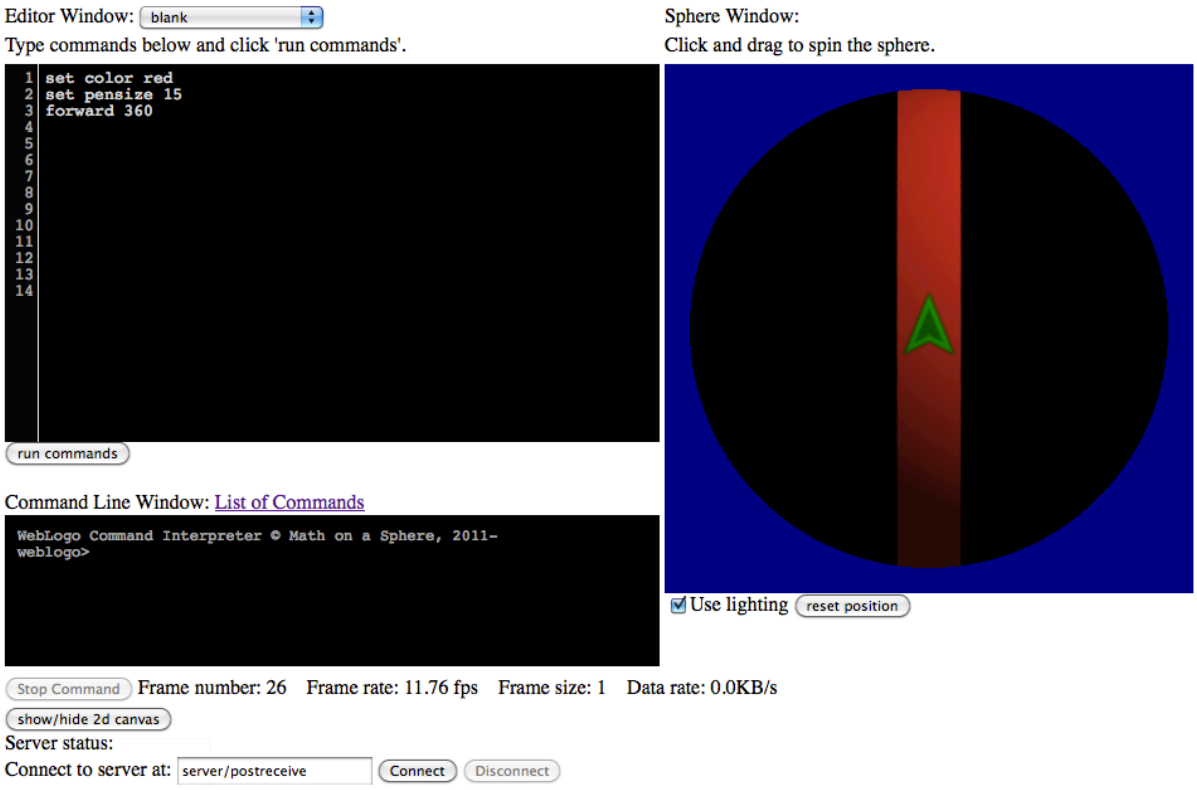


Figure 2.2: This is a screenshot of the user interface as it would be viewed in a browser. On the top left is the editor window: this is where users can compose multiple lines of code to execute together. On the bottom left is the command prompt: this is where single commands can be executed one at a time. To the right is the preview window: this is a preview of how the Science on a Sphere would look when the program is executed, with the green arrowhead representing our version of the "turtle". This preview can serve as a user's own private "spherical display" in the absence of a Science on a Sphere installation.

Just to the right of the editor window is the preview window. This is an adaptation of lesson 11 from a series of handy WebGL tutorials available online[15]. The rasterized ECE images are taken from a hidden 2D HTML canvas element and converted into a texture to be wrapped around the sphere. This window also features 3D lighting effects and manipulation of the sphere's orientation with a simple click-and-drag of the mouse.

There are several buttons available on the user's browser interface as well. The "`run commands`" button just below the editor window allows the user to execute the commands listed in the editor window. Another button just below the command-line interpreter labeled "`stop command`" attempts to gracefully terminate the commands that are currently executing. Just below that is the "`show/hide 2d canvas`" button, which shows (or hides) the 2D HTML canvas element in which the ECE images are rasterized(Figure 2.3). Not only is this view useful for debugging purposes, but it's also an interesting display in its own. Below the preview window is a check box that will allow the enabling/disabling of lighting effects in the preview window. This is useful in some cases where it may be difficult to see what is being drawn to the sphere due to the shadows around the sphere. Next to that, is the "`reset position`" button. Since the view of the sphere preview can be manipulated, this button resets the orientation of the sphere preview to its default position and rotation.

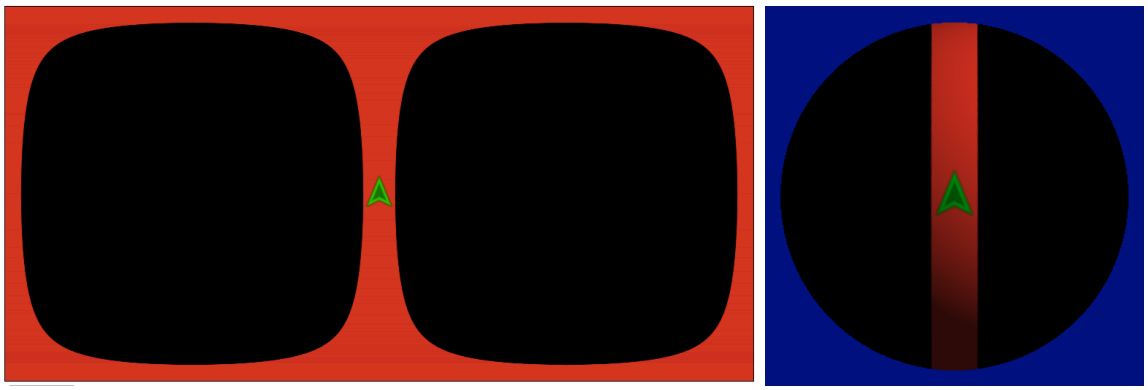


Figure 2.3: To the left is the Equatorial Cylindrical Equidistant (ECE) projection of the image generated for the example in 2.2. For reference, the image on the right shows the ECE image wrapped around the 3D model of a sphere.

At the very top of the page, is a pull down menu populated with a list of WebLogo program samples. Selecting one of these will populate the editor window with the program selected so that it can be executed and viewed by the user as they learn to use the program. This menu could be loaded with "lessons" aimed at teaching progressively more complex programming concepts in WebLogo. A recent addition to this menu is a "my design" option. This selection is tied into a

history record of what the user previously typed in the editor window. That way if the user wishes to swap between their program and one of the sample programs, they can do so without losing progress on their program.

At the very bottom of the user interface is a series of buttons and informational displays pertaining to the MOS server connection. This is how images are sent to the SOS system to be displayed on the sphere. The data displayed includes the frame number, frame rate, frame size, and data rate of the images being sent to the server.

2.1.2 Command Interpreter

The command interpretation is completed in several stages. First, the parser converts the user's program into a parse tree. That parse tree is then traversed by our "transpiler" to generate a stack of base WebLogo commands. Finally, the base command stack is interpreted by a second, simple parser, to determine the rasterization steps needed to make the graphic updates. Each of these command interpretation stages are described in more detail below.

WebLogo Parser: The WebLogo language is defined using a Jison grammar description file. Jison is a `compiler-compiler`, or a parser generator, very similar to Bison (but for JavaScript) and includes its own lexical analyzer modeled after Flex[6]. In other words, it takes a context-free grammar description and generates the parser for that language.

The parser is capable of taking commands from the editor window or from the command-line interface, and generating a parse tree that represents the structure of the program written by the user. Each node of the parse tree has a structure in itself conveying information about that particular construct in the language. At a minimum this includes a `type`, a `handler`, and a `value`. The type tells us what construct in the grammar was used. This information is vital during the development process for debugging, but does not actually affect the outcome of the program. The handler does affect the outcome of the program, and tells us which handler to use when "transpiling" this node. And then of course the value holds the value of the node, which in many cases is a link to the next node in the hierarchy. Additional information may be included

such as args, id, block, condition, etc.

Transpiler/Transcompiler: The "transpiler" or transcompiler is a relatively new term, but the concept behind it is not. A transpiler takes the source code of one language and translates it into the source code of another language. In this case we are translating WebLogo into JavaScript. CoffeeScript is an existing transpiler out there that translates to JavaScript[2]. However, we want to be able to specify our own grammar and syntax so that changes can be easily made later in development, in case studies show certain constructs are easier for children (or learning adults) to understand.

The WebLogo transpiler converts the user's code into a JavaScript program which, when executed, generates a stack of base commands. It does this by recursively traversing the parse tree nodes, and calling the appropriate handler for each one. Handlers are specialized to process a particular node type, returning snippets of a JavaScript program. For example, lets say the user enters the command `forward 90`. The transpiler would generate a program that adds the string "forward 90" to the command stack. Let's look at a slightly more complicated example. Let's say the user instead entered the following code.

```
repeat 3 {
        forward 90
}
```

In this case the handler for `repeat` would call the handler for the `forward` command three times, thus generating the following commands in the command stack.

We will take a closer look at the handlers later on when we discuss the WebLogo programming language in more detail in Chapter 3. Once the transpiler creates the full program and is executed,

the base command stack is processed by a secondary parser.

Base Command Analyzer: The command stack generated by the JavaScript program consists entirely of base commands. Base commands are commands implemented directly with an executor. All WebLogo language structures can be equated to a stack of base commands. For a full list see Table A.1 in the Appendix. The base command parser looks up the appropriate executor for each base command in the stack and applies that executor with the associated argument list. The executors call the rasterizer when needed and change the state of the turtle.

2.1.3 Rasterizer and WebGL Preview

The rasterizer is responsible for generating new images to represent the drawing. Antranig Basman provided the geometry toolkit used to perform all rotations and vector calculations needed to describe the state of the sphere drawing. Dr. Basman also provided many of the executors for the base functions needed to update the images as the drawing is completed. Rasterized images in ECE projection format are stored in the hidden 2D HTML canvas element mentioned earlier in this chapter, and WebGL is used to wrap these images around the surface of the preview sphere. WebGL was used for this task because of its use of the GPU.

2.2 Server Side

In addition to the client side application, Math on a Sphere consists of a server side component. This servlet is responsible for the images displaying on the SOS hardware. It does this by first establishing a connection with the SOS system hardware via a Telnet connection protocol using port 2468[21]. Once the connection is established, the MOS server can direct the SOS system to play images from a specified directory.

```
public void init() {
              final String dir = System.getenv("MOS_IMAGES");
              sos = new SOSConnection("localhost", 2468);
              dataQueue = new ArrayBlockingQueue(10,true);
              sos.connect();
              System.out.println(sos.sendCommand("enable"));
              System.out.println(sos.sendCommand("load "+dir));
```

```
            System.out.println(sos.sendCommand("play"));
...
```

This Telnet connection also allows us to track the progress of the SOS system by sending the `get_frame_number` command as it displays each image in the directory.

This image directory is populated with the posts received from the client side, which sends the ECE image frames generated in the hidden 2D HTML canvas. These images are saved to the disk in a directory specified by the environmental variable `MOS_IMAGES`. The writing of images to this directory is managed as a circular buffer, where the image being written is never ahead by more than 50 frames. The servlet uses the Telnet connection to query the SOS system for the last frame number processed. If the server is less than 50 frames ahead of the SOS system, then the next image is written to the directory. If we haven't received a new image from the client, then we write the last image to disk again. This would happen if the program finished executing, in which case we would want to keep displaying the last image received. If the server is not less than 50 frames ahead of the SOS system, we start throwing out images until the SOS system has had time to catch up. This has the effect of "jumpy" animations if the SOS system gets too bogged down.

One thing we discovered while experimenting with the SOS system is that the images in a directory will be cached if there are fewer than a certain number of frames in the directory. With the system at Fiske, we determined that we needed an image buffer of greater than 50 to prevent the SOS system from caching our images and just looping through the first images received. However, this threshold could differ between various installations of Science on a Sphere. It also appears that after longer-term use (ten minutes of annimation), the SOS system starts to cache again regardless of the image buffer size.

Chapter 3

Programming Language

3.1 The Logo Programing Language

> "Logo is the name for a philosophy of education and a continually evolving family of programming languages that aid in its realization." - Harold Abelson, Apple Logo, 1982.

3.1.1 A Brief History

In the mid 1960s Seymour Papert co-founded the MIT Artificial Intelligence Laboratory with Marvin Minsky[13]. Papert worked with the team from Bolt, Beranek and Newman, led by Wallace Feurzeig, that created the first version of Logo in 1967 [13]. Widespread use of Logo began with the advent of personal computers during the late 1970s[13]. Logo programming activities are found in mathematics, language, music, robotics, telecommunications, and science [13, 12].

The most popular Logo environments have involved the Turtle, originally a robotic creature that sat on the floor and could be directed to move around by typing commands at the computer[13]. Soon the Turtle migrated to the computer graphics screen where it is used to draw shapes, designs, and pictures [13].

Other versions of Logo followed such as TILOGO (released by Texas Instruments), StarLogo (a massively parallel version of Logo) and LEGO TC Logo (a commercial success which reached thousands of teachers and their students) [13].

3.1.2 A Language Designed for Learning

Designed as a tool for learning, the Logo programming language features interactivity, modularity, and flexibility. Logo is generally an interactive language, giving the user immediate feedback on each instruction. This provides the user with valuable information when trying to debug their program[13]. For example, if the user makes a typo when entering the command `forward 90`

```
foward 90
> I don't know how to foward
```

Modularity is achieved through the use of procedures, allowing the user to construct reusable sections of code, a key concept in learning to program. Logo allows procedures to be defined using special words `to` and `end`.

```
to triangle
repeat 3 [forward 90 right 90]
end
```

You'll notice that this segment of code does not require much punctuation or even return line characters at the end of a statement. Logo knows that `forward 90` and `right 90` are separate commands without the user providing more information. This adds flexibility to the language and can prevent frustration, since the targeted users are typically trying to learn other concepts like mathematics in addition to learning how to use a programming language.

Another aspect in which Logo is flexible is in data typing. Many programming languages require the user to specify what data type they want to use. This makes things easier on the computer, but new programmers don't usually think in those terms. Logo allows its users to program without specifying data types [13].

```
print 3 + 4
> 7
print word "3 "4
> 34
```

For WebLogo we wanted to present a similar appearance as Logo, including insensitivity to whitespace, the lack of line terminators and data type specifiers, while still providing the modularity

of functions and various control flow constructs.

3.2 Grammar Specification

WebLogo is defined by a context-free grammar written in Jison. Jison is a compiler-compiler, or a parser generator, very similar to Bison (but for JavaScript) and includes its own lexical analyzer modeled after Flex[6]. Defining the language as a context-free language allows the program to be interpreted into a parse tree for further processing. Having this additional form of the program increases the ability to debug the program, and is a more standard approach to representing a program, which can be used further down the line by extensions such as syntax support and auto-complete.

By default, Bison (and by extension, Jison) constructs LALR(1) parser tables, which do not possess the full language recognition power as LR[7, 8]. The limitations of LALR(1) result in difficulties such as reduce/reduce conflicts when the grammar seems to be perfectly fine[9]. Take a look at the following example[14]:

```
<def> -> <param_spec> <return_spec>
<param_spec> -> <type>
        | <name_list> : <type>
<return_spec> -> <type>
        | <name> : <type>
<type> -> ID
<name> -> ID
<name_list> -> <name>
        | <name> , <name_list>
```

It would seem this grammar could be parsed with only a single lookahead. When a `<param_spec>` is being read, an `ID` is a `<name>` if a comma or colon follows, or a `<type>` if another `ID` follows, making this grammar LR(1). However, there are two contexts in this grammar: the `<type>` at the beginning of `<param_spec>` and at the beginning of `<return_spec>`. They are similar enough that Bison assumes they are the same, due to the same set of rules being activated: the rule for reducing to a name and that for reducing to a type. So Bison makes a single parser state for them both. Combining the two contexts causes the reduce/reduce conflict later. This occurrence means

that the grammar is not LALR(1)[14].

However, there is a workaround for this conflict. One simply has to reform the grammar rule such that they do not look too similar. The Bison manual suggests adding a rule with a `BOGUS` token that is never used, because this introduces the possibility of an additional active rule. For example, in the context after the `ID` at the beginning of `<return_spec>`.

```
<return_spec> -> <type>
        | <name> : <type>
        /* This rule is never used.  */
        | ID BOGUS
```

In my case, I was able to resolve my conflicts by adding another rule that created a new abstraction, as in the case with our expressions. However, these conflicts are noted as difficult to correct and even more difficult to detect[14]. Bison can be set to use a different parser table construction algorithm (such as IELR or canonical LR[7, 8]), so perhaps Jison can as well. However, these alternate algorithms are still experimental, and we were able to resolve our conflicts without resorting to a different algorithm.

3.2.1 Lexemes and Tokens

Lexemes consist of identifiers, literals (numbers, strings, and boolean values), special words and operators, while a token is a category given to a lexeme [23]. The token category will be represented in all capital letters if it takes the form of a terminal node in the grammar.

A WebLogo `IDENTIFIER` is defined as starting with a roman letter, followed by any number of roman letters, digits, or dots. The dot is allowed to be part of an identifier to allow the implementation of members and member functions. WebLogo currently only uses this to recognize JavaScript math functions like `Math.cos` and `Math.sqrt`. A `NUMBER` in WebLogo consist of digits and can contain or start with a decimal point. A `NUMBER` may also be represented in scientific notation using an upper- or lower-case `e`. A `STRING` must be surrounded by double quotes and must not contain single or double quotes, and a `BOOLEAN` may be represented by the strings `true` and `false`.

Special words are identifiers that hold special meaning to the programming language. WebLogo's special words consist of control-flow words and commands. Control-flow words consist of grammar constructs like `repeat`, `function`, and `if`, while commands are built in functions like the familiar `forward`, `right`, and `left` used in the Logo language. See the appendix for a full list of WebLogo's commands.

The mathematical operators `(+-/*^=)` and the logical operators (==<>) are all binary operators recognized by WebLogo. The only unary operator currently recognized is unary minus.

3.2.2 Values and Arrays

Values consist of numbers, strings, identifiers, booleans, and empty parentheses. Using the Buckus-Naur Form, we define these as values. The < > brackets are abstractions used to represent syntax structures[23], while the bar indicates that an abstraction can take on any one of the values listed. For example, the `<value>` abstraction can take on the value of `<number>`, `<string>`, `<identifier>`, `<boolean>` or a set of empty round brackets.

```
<value> -> <number>
        | <string>
        | <idendifier>
        | <boolean>
        | ( )
```

An array is defined using the following grammar rules, and consists of any number of sequential values separated by commas and enclosed by square brackets.

```
<array> -> [ ]
        | [ <element_list> ]
<element_list> -> <expr>
        | <expr> , <element_list>
```

This is an example of a recursive rule, where the LHS (left hand side) of the rule also appears in the RHS (right hand side) of the rule. Listing the recursive call on the right-most side of the RHS makes this rule part of an LR grammar, meaning the tokens are processed left-to-right. These two grammar rules will allow an array of any number of values separated by commas and enclosed with square brackets to be parsed.

3.2.3 Expressions, Restricted Expression, and Logic Expressions

Expressions are constructs that need to be evaluated. For example, a mathematical expression, or a boolean conditional. These are constructs that can be condensed to a single value representation. In WebLogo, expressions, restricted expressions, and logic expressions are treated separately for a couple of reasons. The main reason is to work around the issue mentioned before about Jison needing a LALR(1) grammar definition. Another reason is to define the grammar in such a way that we can restrict the conditions in which certain expression types may be used. For example, we only want to allow a logic expression to be used in a conditional assessment. This also helps improve the readability of the grammar.

The all inclusive expression rule is what we are defining as simply "expression" or `<expr>`. This includes all types of expressions, including the unary minus operator and logical expressions. The separation of the unary minus is necessary for disambiguation with the binary minus operator.

```
<expr> -> <re>
        | - <re>
        | <logic_expr>
```

Restricted expressions `<re>` include all expressions except the unary minus operator and logic expressions. This mostly includes mathematical expressions, but also includes the evaluation of a function call.

```
<re> -> <re> MATH_OP <re>
        | ( <re> )
        | ( <function_call> )
        | <val>
```

where a `MATH_OP` is defined as any of the five mathematical operators `(+-*/^)`. The logic expression, as the name suggests, only includes expressions that can be evaluated to a boolean value.

```
<logic_expr> -> <re> LOGIC_OP <re>
```

3.2.4 Statements

While expressions need to be evaluated, statements actually do something with those evaluations. In the WebLogo language that includes assigning a value to a variable (or a function), a function call, a `repeat` statement, or an `if` statement.

```
<stmt> -> <assign>
        | <function>
        | <repeat_stmt>
        | <if_stmt>
```

3.2.5 Variable Assignments, Function Declarations, and Parameter Lists

Variable assignments and function declarations are treated as the same entity at the grammar level. We explain this in more detail later on in the section on handlers (see section **??**).

```
<assign> -> <identifier> = <expr>
        | <identifier> = FUNCTION ( ) <block>
        | <identifier> = FUNCTION <param_list> <block>
<param_list> -> <array>
        | ( <array> )
```

The parameter list abstraction `<param_list>` consists of any array construct and may optionally be surrounded by round brackets.

3.2.6 Function Calls

Function calls in WebLogo automatically treat the following token as its argument unless it is a built-in function which takes no parameters, or a `<builtin_null>`. To pass multiple arguments into a function, one simply passes them in the form of an array. To pass no arguments, one uses an empty array or empty parentheses.

```
<function_call> -> <val> <arguments>
        | ( <function_call ) <arguments>
        | <builtin_null>
        | <set_stmt>

<arguments> -> ( <expr> )
        | <val>
        | <array>
```

The exception is with built-in null functions and set statements. Built-in null functions can be recognize purely by their identifier and thus the command may be used with no empty brackets or parentheses. The list of built-in null functions can be found in Table A.1.

The set statement follows a slightly different syntax format, and is meant to increase the flexibility in programming styles. The set statement always has two arguments, a property name and a value for that property. Since we know the number of arguments ahead of time, the arguments need only be separated by a space.

```
<set_stmt> -> SET <val> <expr>
```

Providing this additional format was intended to give feedback on what programming constructs are easier for new programmers learn.

3.2.7 Repeat Statements

The repeat construct works the same way as it does in Logo with a slight variation in the syntax, using curly braces instead of square brackets. We specify the repeat construct using the following rule.

```
<repeat_stmt> -> REPEAT <expr> <block>
```

The REPEAT token will recognize `repeat` or `REPEAT`. The following would be a valid `repeat` statement in `WebLogo`.

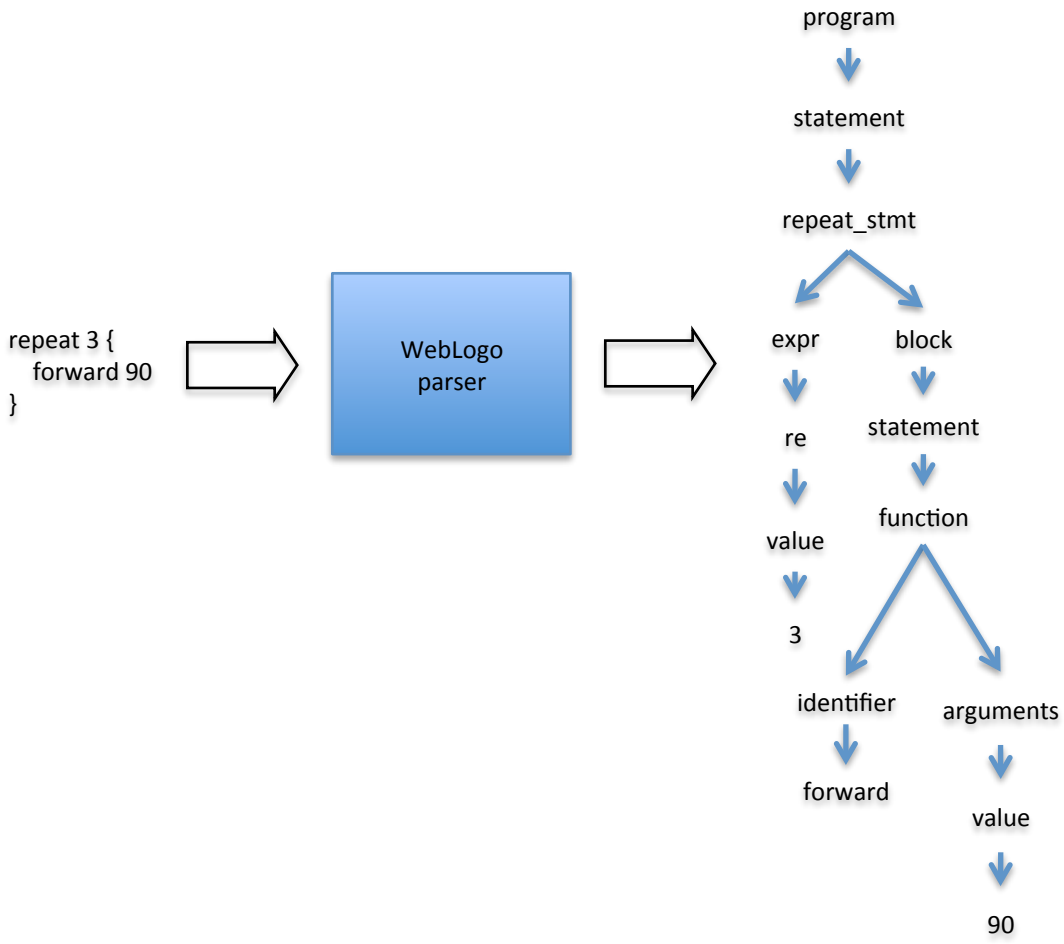
```
repeat 3 {
        forward 90
        right 90
}
```

On a 2D surface this would draw three sides of a square, but on the sphere it draws a triangle with three right angles. In fact, the `expr` abstraction will allow `repeat` to take any valid expression as the repeat specifier. For example, the following program is equivalent to the previous example.

```
N = 4
repeat (N-1) {
        forward 90
        right 90
}
```

Within the repeat construct there is not an obvious way to determine how many times the interior has executed as there would be in modern languages. For the sake of keeping the code easy to read, I chose to leave out additional syntax.

Figure 3.1: The user's input program on the left is parsed by the WebLogo parser generated from our context-free grammar. The resulting parse tree is to the right.



3.2.8 If Statements

`WebLogo` supports `if` and `if-else` statements with the same syntax as JavaScript. However, `if-elseif-else` constructs are not currently supported.

```
<if_stmt> -> IF <logic_expr> <block>
        | IF <logic_expr> <block> ELSE <block>
```

3.2.9 Reference Structures

The rest of the rules are here mainly to aid in the task of debugging by making the parse tree reflect the full path of rules used in the context-free grammar. However, some of these rules also help group high-level sections of code. For example, the `<stmts>` rule allows us to have any number of individual statement constructs in a sequence.

```
<stmts> -> <stmt> <stmts>
        | <stmt>
```

The `<block>`, referenced in several of our rules already, is simply a set of statements enclosed in curly brackets.

```
<block> -> { <stmts> }
```

The final connections in the hierarchy are achieved with the program and weblogo constructs.

```
<weblogo> -> <block>
        | <stmts>

<program> -> <weblogo> EOF
```

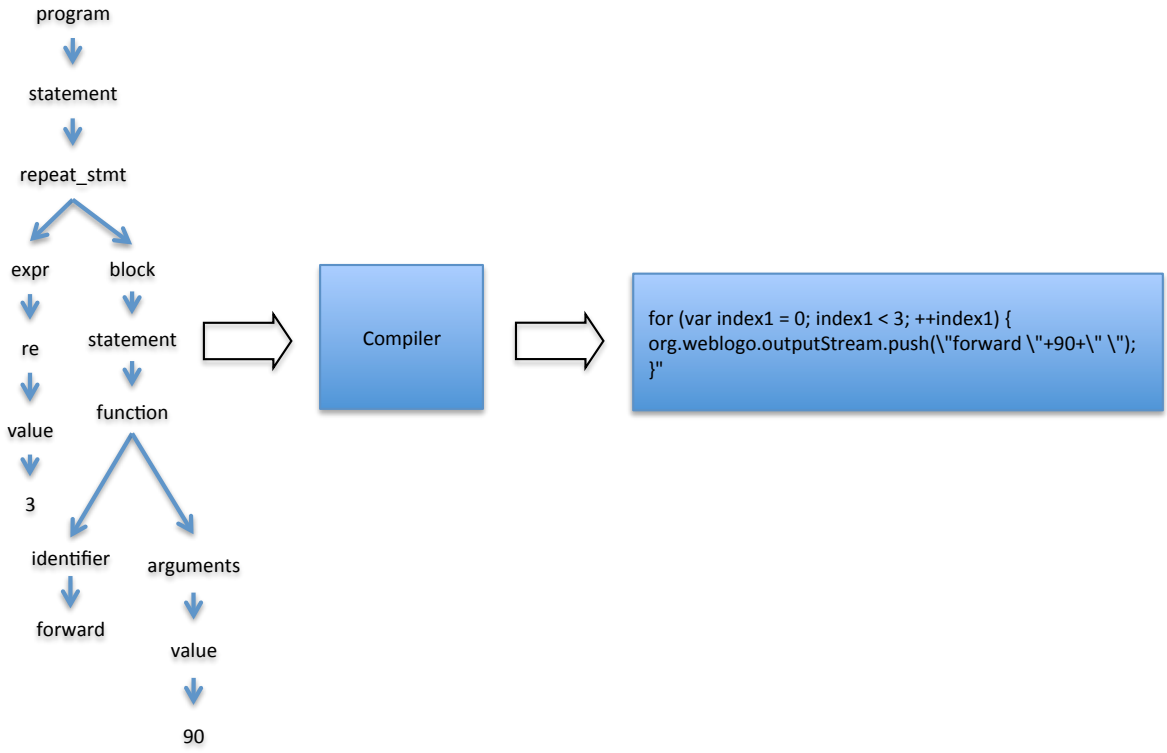
In the end, we end up with a parser that will return a parse tree corresponding to our context-free grammar, as with the `repeat` example (Figure 3.1). From there, the parse tree is "transpiled" into a JavaScript program (Figure 3.2).

3.3 Translator and Node Handlers

The parse tree generated by the parser is converted into a JavaScript program that will build the base command stack. The translator does this by recursively traversing the nodes of the parse tree and calling the referenced node handler for each node. Let's take a closer look at the repeat statement example covered in the overview section.

The repeat node has two child nodes: the `<expr>` node and the `<block>` node. The handler for the `<repeat>` node knows that `<expr>` will tell it how many times to repeat the statements

Figure 3.2: The parse tree is transpiled into a JavaScript program that will build a stack of base commands.



listed in the `<block>`. So it calls the compiler on these two child nodes and constructs a repeat statement in JavaScript, returning this new JavaScript snippet to its parent node. So when the compiler is called on the `<expr>` node, the tree is traversed all the way down to the `<value>` node, which determines that the value "3" should be returned, giving the repeat handler the number of times to repeat the statements included in the `<block>` node. When the compiler is called on the `<block>` node, it traverses down to process the `<function>` node.

By visiting each node in this fashion, the handlers piece together larger components of the JavaScript code. Taking a look at the repeat handler below, we can see that the handler constructs a string by processing the child nodes separately, folding the results into the `value` field of the returned node for the repeat handler.

```
org.weblogo.nodeHandlers.repeat_stmt = function(node, program, compiler) {
        var index = ++compiler.depth;
        var repeatN = compiler(node.args[0], "").value;
        node.value = "for (var index"+index+" = 0; index"+index+" < "+repeatN+";
                          ++index"+index+") ";
        node.value += compiler(node.args[1], "");
        return node;
}
...
org.weblogo.nodeHandlers.statement = function(node, program, compiler) {
       node.value = compiler(node.value, "").value;
        if(node.type === "statement") {
                node.value += ";\n";
        }
    return program += node.value;
}
```

Now let's take a look at the handler for the <function> node. Functions in WebLogo can be built-in or user-defined. The same handler is used for both, allowing the context-free grammar to treat them as identical entities (with the exception of built-in functions with null parameters, see section 3.2.6). The handler traverses the <identifier> node to determine that we are using the function "forward", and a traversal of the <arguments> node tells the handler to pass the value "90" as a parameter to "forward". How the function handler proceeds from here depends on whether the function being used is a WebLogo built-in function or a user-defined function. Since the function being used is a built-in function (or "base" function), this is simply a matter of building a string with the base function name an parameters, resulting in "forward 90". A JavaScript snippet is generated that will push this base command string to the Base Command Stack.

```
"org.weblogo.outputStream.push(\"forward\"+90+\"\");"
```

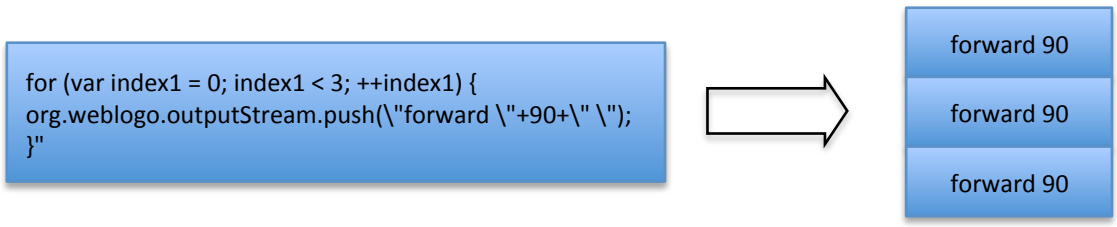
If we were using a user-defined function instead of the base command "forward", then the function handler would instead generate the JavaScript snippet to call that user function with the parameters used. Since the user function would have beed previously defined in the form of another JavaScript snippet, one that would push the necessary base commands to the stack, this works perfectly.

After the full JavaScript translation is completed, it is executed, generating the Base Command Stack for further processing.

3.4 Base Command Stack

The Base Command Stack will consist of only base commands as listed in Table A.1. These are commands such as “forward”, “right”, and “penup”. This command stack is processed by a second parser. This parser is much simpler, tokenizing on spaces and expecting the first token to be the command name, with each following token being a parameter value. Each command is mapped to its executor, responsible for the geometric calculations and rotation necessary to carry out its task. For example, “forward” maps to the line executor, which handles drawing the line in the direction of the turtle’s current heading for the number of steps specified as its parameter. It also handles animating this process by measuring the number of “ticks” that have passed, generating as many images as needed to take up the appropriate amount of time.

Figure 3.3: The base command stack as generated by the JavaScript program to the left.



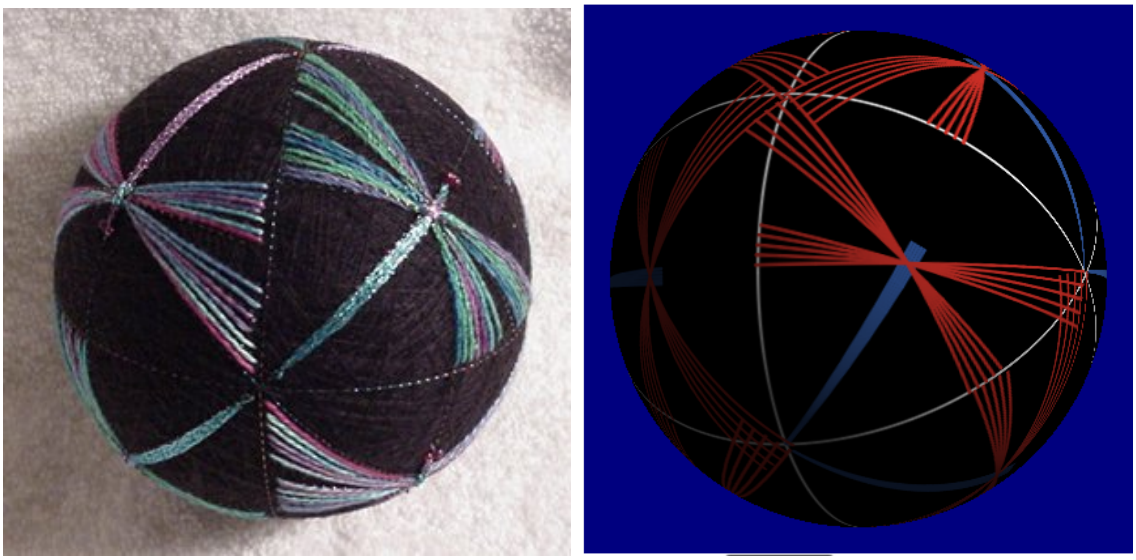
Chapter 4

Math on a Sphere in Action

4.1 Temari Patterns

One interesting use of the MoS system is for the digital creation of "Temari", a folk art originating from China and passed on to Japan. Traditionally, these forms of geometric art are created by hand with a styrofoam core and some yarn [20]. Inspired by Karen McPoyle's dragonfly pattern at TemariKai.com, I decided to try creating my own variation of the design using MOS.

Figure 4.1: On the left is the original temari pattern 99 KM07 by Karen McPoyle, Pennsylvania. On the right is my own translation of this pattern into a WebLogo program.

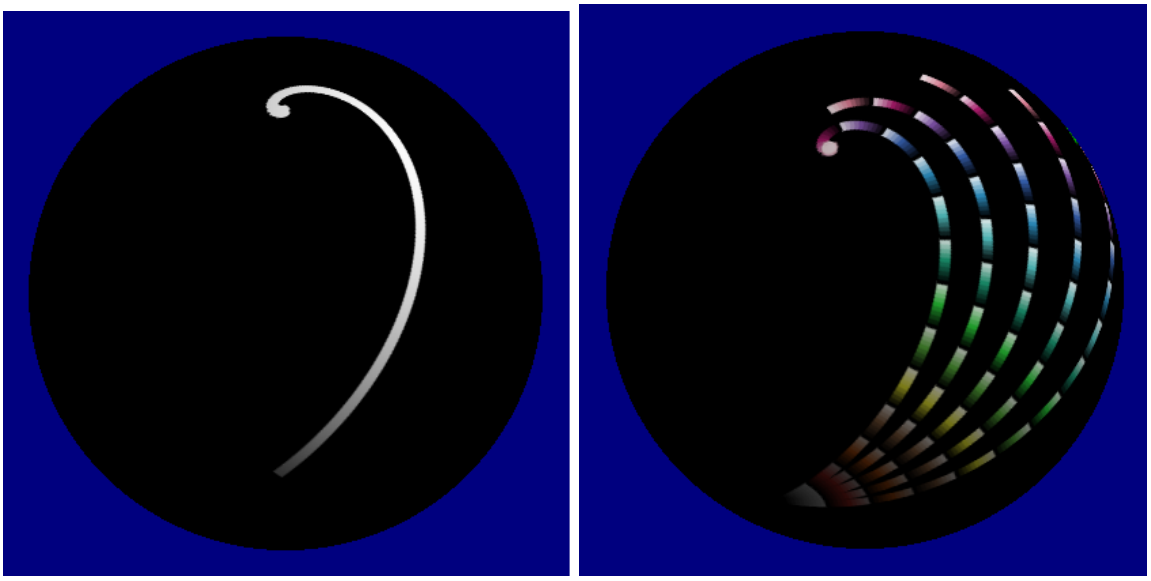


4.2 Spirals and Curves

The WebLogo programming language allows you to draw along great circles and small circles, but you could also draw spirals. Defining a function that updates the heading with each step the turtle takes could, in theory, describe any curve you wish. Take a look at the spiral function defined below. This function sets the heading to `angle` with every step, causing the turtle to spiral in on the pole (Figure 4.2).

```
spiral = function [steps, angle] {
        repeat steps {
                setheading angle
                forward 1
        }
}
```

Figure 4.2: On the left is a single spiral generated with the spiral function described above with an angle of 45 degrees. On the right is the same spiral with an increasing angle parameter and an additional loop to cycle through the color codes.

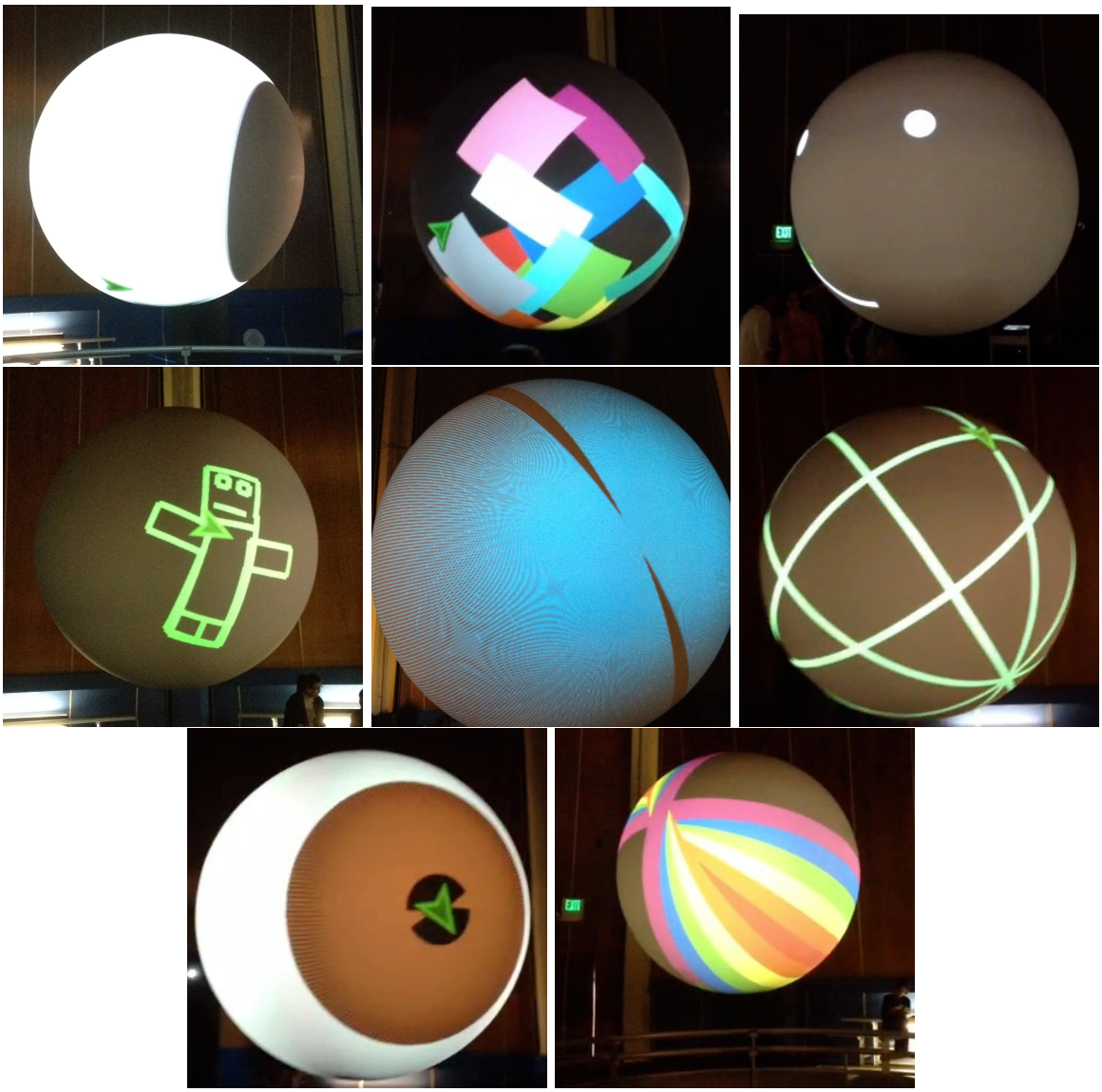


4.3 Visit to Berkeley

In late February of 2012, Antranig Basman and I took a trip to University of California at Berkeley to help setup the Math on a Sphere system on their SoS installation. Dr. Sherry Hsi of the Lawrence Hall of Science conducted a workshop with a dozen or more kids working with the software. We were there to provide technical support and work out a few last minute bugs during the workshop.

After an hour or so of introductory activities, the kids got started on programming, and most did quite well. Many even requested more time to work on their projects when their time was up. One student came up with the creative idea to turn the sphere into a giant eyeball. Combined with the iPad's ability to rotate and tilt the image this was quite convincing. One kid would rotate the image position to make the eye "look" at the other kids as they walked around the sphere.

Figure 4.3: These are images taken at the Lawrence Hall of Science from programs created by the kids who participated in the workshop.



Chapter 5

Future Work

This thesis implemented a new programming language for the Math on a Sphere project, supporting nested functions, proper scoping, mathematical expressions, and conditionals. The web-based approach makes this software accessible to all, and provides users with a way to experiment with their programs even in the absence of a sphere. With a two-year time frame planned for this project, it is expected that further development will continue. Some of the improvements listed here are minor while others are a more significant undertaking.

5.1 The Language

Additional work on the language could include implementing commands that would hide or show the turtle, extending the if-else statements to include if-elseif-else constructs, include statements, and support for return values. The current structure makes it difficult to return a value for base commands since those commands are not interpreted until much further down the line, so a creative solution would be needed here. Implementing an include statement would probably require the use of Ajax to read stored files, since JavaScript does not have its own include construct. This would allow user to import functions and programs shared by their peers.

5.2 User Interface

Changes to the user interface would probably start with implementing the CodeMirror support for WebLogo. One of the major obstacles for kids using the MOS interface centered around

the ability to detect typos, or to simply type. It was thought that providing the alias commands (listed in Table A.1) would help with this, however they still had difficulty detecting a typo once it had been made. Implementing the CodeMirror support will give us tools for syntax checking and even autocompletion. This could be vital, since our grammar type makes error messages difficult to report. If CodeMirror is not enough, there is the option to add buttons or other constructs like the ones used in Scratch[3] or in the original undergraduate prototype project[18].

5.3 Social Programming

A future goal of the two-year Math on a Sphere project is to implement some form of social interface for MOS programming. This could take the form of a wiki, where users upload, share and talk about their programs. This wiki interface could be used to present lesson plans for students, giving teachers the tools to present assignments for the sphere.

5.4 The Future of Math on a Sphere

Math on a Sphere shows great promise in its ability to engage the public. It is already being used successfully by children 9-13 years of ages, which will provide feedback on ways to make this program even more accessible to learners. As further development continues in the next year, it is easy to imagine seeing this interface pop up at SOS installations all around the sphere we live on. What will this little turtle see as it peeks over the horizon? With the continued support of the those dedicated to the success of this project, it should be a splendid sight indeed.

Bibliography

[1] Harold Abelson and Andrea diSessa. Turtle Geometry: The Computer as a Medium for Exploring Mathematics. MIT Press, 1980.

[2] Jeremy Ashkenas. Coffeescript. <http://coffeescript.org/>.

[3] Lifelong Kindergarten Group at the MIT Media Lab. Scratch. http://scratch.mit.edu/.

[4] Bell, Lewenstein, Shouse, and Feder. (Editors). Learning science in informal environments: People, places, and pursuits. National Research Council of the National Academies, 2009.

[5] Burbules and Reese. Teaching logic to children: An exploratory study of Rocky's Boots. assessing the cognitive consequences of computer environments for learning (acccel). California Univ., Berkeley. Lawrence Hall of Science., 1984.

[6] Zach Carter. Jison: Your friendly javascript parser generator!, 2010. MIT Licensed website. <http://zaach.github.com/jison/>.

[7] Joel E. Denny and Brian A. Malloy. Ielr(1): Practical lr(1) parser tables for non-lr(1) grammars with conflict resolution. Proceedings of the 2008 ACM Symposium on Applied Computing, ACM, New York, NY, USA, pp. 240245., 2008.

[8] Joel E. Denny and Brian A. Malloy. The ielr(1) algorithm for generating minimal lr(1) parser tables for non-lr(1) grammars with conflict resolution. Science of Computer Programming, Vol. 75, Issue 11, pp. 943979., November 2010.

[9] Frank DeRemer and Thomas Pennello. Efficient computation of lalr(1) look-ahead sets. ACM Transactions on Programming Languages and Systems, Vol. 4, No. 4, pp. 615649., October 1982.

[10] Eisenberg. Computational diversions: Turtle really and truly escapes the plane. International Journal of Computers for Mathematical Learning, 15:7379, 2010.

[11] Eisenberg and Hsi. Math on a sphere: An interactive exploration of 3d surfaces for public audiences. Proposal for National Science Foundation (NSF) Informal Science Education solicitation num. 10-565, project type: Pathways, 2010.

[12] W Feurzeig. Programming-languages as a conceptual framework for teaching mathematics. sigcue bulletin, 1970.

[13] Logo Foundation. Logo foundation, 2012. <http://el.media.mit.edu/logo-foundation/index.html>.

[14] Inc. Free Software Foundation. Mysterious conflicts - bison 2.5, 2007. <http://www.gnu.org/software/bison/manual/>.

[15] Giles. Learning webgl: lessons 'n' links. <Learningwebgl.com >.

[16] Marijn Haverbeke. Codemirror. <http://codemirror.net/>.

[17] Randi Korn and Inc. Associates. Remedial evaluation of the science on a sphere exhibition, Feb 2010.

[18] MacFerrin, Shulman, Robbins, Hallesy, Bailey, and Shelton. Lasermission. University of Colorado Boulder Undergraduate Senior Project <http://code.google.com/p/lasermission/>.

[19] McNeil, Grandau, Knuth, Alibali, Stephens, Hattikudur, and Krill. Middle-school students understanding of the equal sign: The books they read cant help. COGNITION AND INSTRUCTION, 24(3), 367385, 2006.

[20] Karen McPoyle. Temari pattern 99 km07. <http://www.temarikai.com/patterns/tem99km07.html>.

[21] NOAA. Science on a sphere, 2012. <http://sos.noaa.gov>.

[22] Dave Schindler, 2010. <http://terminal.jcubic.pl/>.

[23] Robert W. Sebesta. Concepts of Programming Languages. Addison-Wesley, 2004.

Appendix A

WebLogo Tabels

This appendix contains supplemental information about the `WebLogo` programming language that has been referenced throughout this paper. First you will find a list of the base commands along with aliases, parameter specifications, and descriptions. Then the properties available for the `set` command, which uses a different syntax structure from `WebLogo` function constructs. Finally, you will find a description of the color codes and color string mappings.

Table A.1: Here is a list of the commands used in the WebLogo language along with their descriptions.

command	alias	parameters	description
clearall	ca	builtin_null	Reset parameters to their default values and clear the drawing.
cleardrawing	cd	builtin_null	Clear the drawing.
forward	fd	x	Move forward along the current heading for x ticks, ticks represent degrees around the sphere.
back	bk	x	Move back x ticks, ticks represent degrees around the sphere.
right	rt	x	Adjust heading right by x degrees.
left	lt	x	Adjust heading left by x degrees.
penup	pu	builtin_null	Moving the turtle will not draw a line after issuing this command.
pendown	pd	builtin_null	Moving the turtle will draw a line after issuing this command.
setheading	sh	x	A heading in degrees from 0 - 180 where 0 is the initial direction of the "north" pole from the origin [0,0]. As the turtle passes through the pole, the heading remains the same. A heading of 180 would represent the same path but moving in the opposite direction.
getheading	gh	builtin_null	Print the current heading in the "terminal window".
setposition	sp	[x,y]	Set the position of the turtle (without drawing a line to get there). Spherical coordinates [x,y] are used to describe the turtle's position on the surface of the sphere, where x is measured in degrees from the equator to the north and y is measured from the origin along the equator in an eastward direction. The origin [0,0] is at the point along the equatorial line closest to the viewer in the "sphere preview window".
getposition	gp	builtin_null	Print the turtle's current position to the "terminal window".
setrotationaxis	sra	[x,y]	Sets the rotational axis used when moving the turtle along the surface of the sphere. This allows the drawing of a small circle, or latitude lines. Spherical coordinates [x,y] and the center of the sphere represent the axis of rotation.
print		x	Print x to the "terminal window". The parameter x can be a string or variable identifier, but does not detect variables within a string.
set		x y	Sets the specified property (x) with the given value (y). This construct uses a different syntax, not requiring the square bracket notation used for function calls. This syntax always requires a property and value specifier. See the list of properties in the following table.

Table A.2: Here is a list of the properties used in the WebLogo language. These are set using the "set" command described in the previous table.

Property Name	parameter	description
pensize	x	integer representing the pixel width of pen
color	x	sets the drawing color to the given enumerated color strings or numeric color value

Table A.3: The numeric color system ranges from 0 to 139. Values higher then 139 can be used but the list of unique colors ends at 139. Black is color value 0 and white is color value 9.9 with a gradient of grays between the two. The next interval of 10 gives a gradient of reds, with 10 being the darkest red available and 19.9 being the lightest. Given this color coding scheme, "pure" colors are approximated with color codes ending in 5. Here is a list of color strings mapped to their color codes as they are implemented in the WebLogo language.

color name	numeric code
black	0
gray	5
white	9.9
red	15
orange	25
brown	35
yellow	45
green	55
lime	65
turquoise	75
cyan	85
sky	95
blue	105
violet	115
magenta	125
pink	135

Appendix B

WebLogo Sample Programs

This appendix contains examples of WebLogo programs referenced in this paper. These programs are also used in the pulldown menu on the user interface.

Listing B.1: dragonfly.wbl

```
set pensize .5
set color white
setspeed (3*PI)
fd 360 rt 90 fd (360+90) rt 90 fd 360

x = 54
body = 60

dragon_tail = function [body_color] {
  rt 43
  set color body_color
  repeat 5 {
    repeat 2 {
      pendown fd body
      penup fd (180-body)
    }
    rt 1
  }
  lt 47
```

```
}

wing = function [wing_color] {
  set color wing_color
  repeat 5 {
    repeat 2 {
      pendown fd (90-x)
      penup fd 90
      pendown fd x
    }
    lt 4
  }
}

dragon = function () {
  dragon_tail blue
  fd 90 rt (90+45) fd x
  wing red
  rt (180-24)
  wing red
}



repeat 4 {
  dragon()
  penup rt 65 fd x rt 45
}
```

Listing B.2: randcircle.wbl

```
circle2 = function [t,p,r] {
  penup setpos [t,p] fd r
  pd right 90
```

```
  setrotationaxis [t,p]
  fd 360
}


repeat 14 {
  rand = (Math.random())
  circle2 [rand*360,rand*360,rand*60]
  set color (rand*139)
  set pensize (rand*50)
}
```

Listing B.3: triangle_size.wbl

```
tri = function [size] {
    r = size*PI/180
    a = (Math.cos(r))
    b = (Math.sin(r))
    c = (a - (a * a)) / (b * b)
    theta = 180 - (Math.acos(c))*180/PI
    repeat (3) {
        forward size
        right theta
    }
}

tri 10
```